



# Building, Deploying and Scaling Web Applications in 2015

ASINT

67622 João Lima [j.lima91@yahoo.com](mailto:j.lima91@yahoo.com)

# Contents

<b>1</b>	<b>Motivation</b>	<b>2</b>
<b>2</b>	<b>Project Overview</b>	<b>2</b>
<b>3</b>	<b>Architecture</b>	<b>2</b>
3.1	Web Application . . . . .	2
3.1.1	Database . . . . .	3
3.1.2	Web Server . . . . .	4
3.1.3	Client . . . . .	7
3.2	Scaling the application . . . . .	7
3.2.1	Environment used . . . . .	7
3.2.2	Services . . . . .	10
3.2.3	Service Registration and Load Balancing the two Web- Server instances . . . . .	13
3.2.4	DNS Registration, Subdomains and High Availability Proxy . . . . .	14
3.2.5	High Level View of the Architecture . . . . .	16
<b>4</b>	<b>Appendix</b>	<b>18</b>
4.1	Helpful links . . . . .	18
4.2	DigitalOcean machines . . . . .	18
4.3	DNS configuration . . . . .	19
4.4	Fleet Files (only a subset) . . . . .	20
4.5	Other Considerations . . . . .	22
4.6	Other tools used . . . . .	22

# 1 Motivation

*Firebase* is a *Platform as a Service* (PAAS) that managed to get really popular in the last year. According to their website: “Firebase can power your app’s backend, including data storage, user authentication, static hosting, and more. Focus on creating extraordinary user experiences. We’ll take care of the rest.”

I used *Firebase* for a personal project and in a short amount of time, without writing a single line of server code, I had a real-time web-application running with persistent data, that could scale automatically, depending on my application’s needs. This project is a result of trying to understand how to implement from scratch some of firebase’s features.

## 2 Project Overview

I decided to build a real-time *Single Page Application* (SPA) from scratch, making use some of the most popular web-developing tools of the last year. The software stack used for this project can be used for real world applications (for example, *Firefox Devtools* is using *React* and *Redux*). Some things like *unit-testing* and *continuous integration* are missing in this project, when compared to real world ones.

I’m going to make a simple upvote/downvote news forum with live comments. The web application makes use of a database, a server/api and the clients. The application is going to be in real time, for example, every time a user upvotes or downvotes a post, all users connected to the website will receive the updated information on the screen, without updating the browser. This is done using *websockets*. Every time there is a change in the posts data inside the database, the server is signaled and then broadcasts the changed information to all the socket-connected clients. The API will be public with the corresponding documentation for it to be consumed.

For scalability the web server and database will have multiple nodes, and all the requests are going to be controlled by a load balancer.

## 3 Architecture

### 3.1 Web Application

For this project I implemented a multi tiered web application with a database, web-server and a client. The diagram is represented on Figure 1. A client starts by logging into our web server. He then receives a JSON web token

with his identification and authorization levels. If for example, a user upvotes or downvotes a post, the client sends a POST method to our web-server. The web-server then issues an update to the corresponding post inside the “posts” table. The web-server has a previously established *change feed* connection with the “posts” table in the database. This means that, every time this table is updated, the server receives a message containing the information that changed. The server is then responsible for broadcasting these changes to every client connected to our web-server.

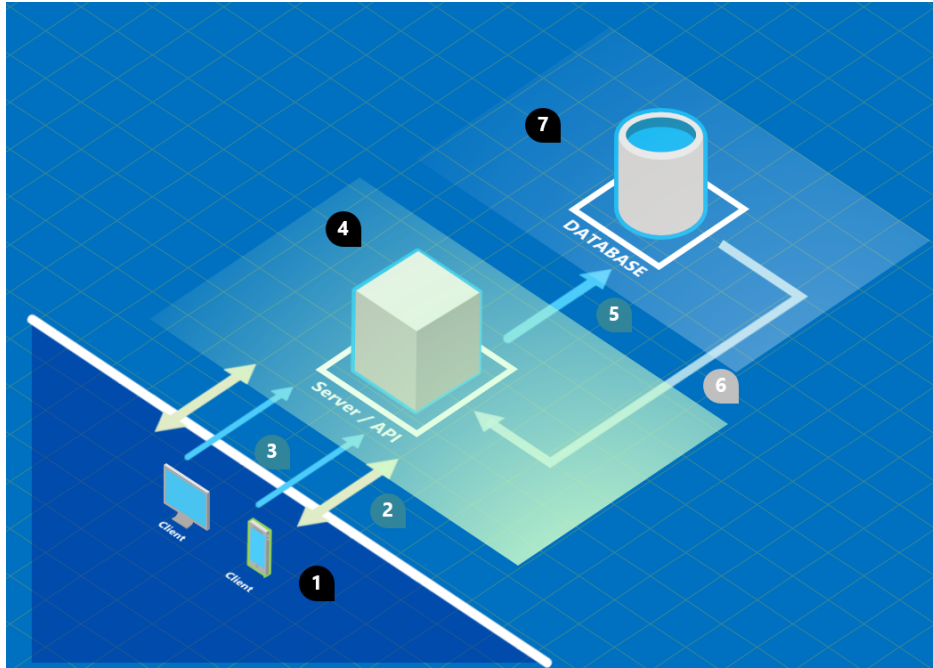


Figure 1: three-tiered web application : 1 - Clients; 2 - Bidirectional *web-socket* connection between client and server is established. This connection is used to send the changes made to the database to the client in real-time; 3 - Client logs into the API and receives a JSON Web Token. Future API requests include the token granting the user authentication. The API is kept stateless; 4 - Web-Server; 5 - Database query; 6 - Unidirectional connection between database and server. Every time the Posts and Comments’ tables are changed, the database signals the server; 7 - Real Time Database.

### 3.1.1 Database

For the database I chose *RethinkDb*, a NoSQL database designed with scalability in mind. It uses the ReQL language for queries and can be used to

build real-time applications using a feature called *changefeeds*, which allows “clients to receive changes on a table, a single document, or even the results from a specific query as they happen.” - <https://rethinkdb.com/docs/changefeeds/javascript/>.

### 3.1.2 Web Server

To implement the web-server, I decided to go with *node.js* and the *express* web framework. To communicate with the database the official javascript driver from *RethinkDb* is used. For broadcasting information to the clients, the *socket.io* library is used. This library uses the WebSocket protocol with polling has a fall back option. After creating the database connection, two changefeeds are opened, one for the “posts” table and the other for the “comments” table (see code below 1).

Then we create a router instance and the endpoints for our REST API. The route definitions can be seen on the block of code 2 below. The API has both protected and unprotected routes. The protected routes require a valid JSON web token, otherwise the server returns the HTTP code 403 with a message saying “No token provided”. To verify the token validity a custom middleware function is used - *verifytoken()*. There are also special routes which require a higher level of authorization (commented as “/\*scoped routes\*/” below).

Listing 1: Broadcasting the changes returned by the changefeed connections

```
/*emit changes from comments*/
r.connect(config.rethinkdb).then(function(conn) {
  this.conn = conn;
  return
    r.table("comments").changes().run(conn).then(function(cursor)
    {
      cursor.each(function(err, item) {
        io.emit('commentstream', item);
      });
    })
  }).error(function(err) { console.log("Failure:", err); })

/*emit changes from posts*/
r.connect(config.rethinkdb).then(function(conn) {
  this.conn = conn;
  return
```

```

        r.table("posts").changes().run(conn).then(function(cursor) {
        cursor.each(function(err, item) {
            io.emit('poststream', item);
        });
    })
}).error(function(err) { console.log("Failure:", err); })

```

---

Listing 2: API endpoints (server.js)

```

var apiRoutes = express.Router();

/*unprotected routes*/
apiRoutes.route('/authenticate').post(authentication);
apiRoutes.route('/posts').get(getPosts);
apiRoutes.route('/users/:login').put(createUser);
apiRoutes.route('/comments/:pid').get(getComments)

/*route middleware to verify token*/
apiRoutes.use(verifytoken);
/*protected routes*/
apiRoutes.route('/comments/:pid').put(createComment);
apiRoutes.route('/posts').post(createPost);
apiRoutes.route('/posts/:pid/upvotes').post(upvotesPost);
apiRoutes.route('/posts/:pid/downvotes').post(downvotesPost);

/*scoped routes*/
/*delete*/
apiRoutes.route('/users/:login').delete(check_scopes(['delete']),deleteUser);
apiRoutes.route('/comments/:pid/:id').delete(check_scopes(['delete']),deleteComment);
/*read*/
apiRoutes.route('/users').get(check_scopes(['read']),getAllUsers)
apiRoutes.route('/users/:login').get(check_scopes(['read']),checkUser);

/*mount apiRoutes with /api prefix*/
app.use('/api',apiRoutes);

```

---

**API Authentication** - To authenticate the API I decided to go with a JSON Web Token library, instead of a full framework like OAuth or OAuth2. A JSON Web Token (jwt) is composed by three parts separated by a “.” (header.payload.signature). An example of a token payload from my API is represented below:

```

{

```

```

    "login": "teste",
    "scopes": [
      "read",
      "delete"
    ],
    "iat": 1450040162,
    "exp": 1450126562
  }

```

This token identifies the user “teste” which has two scopes “read” and “delete” allowing him to access all the routes on my API. For more information about JSON Web tokens, consult [www.jwt.io](http://www.jwt.io).

**API documentation** - I created an interactive web page for my API documentation using the *swagger* framework. To access it, go to <http://docs.lslima.me>

<b>auth</b>	
POST	/authenticate
<b>comments</b>	
GET	/comments/{pid}
PUT	/comments/{pid}
DELETE	/comments/{postId}/{commentId}
<b>posts</b>	
GET	/posts
POST	/posts
POST	/posts/{pid}/upvotes
POST	/posts/{pid}/downvotes
<b>users</b>	
GET	/users
DELETE	/users/{login}
GET	/users/{login}
PUT	/users/{login}

[ BASE URL: /api , API VERSION: 1.0.0 ]

Figure 2: API documentation. API can be directly consulted on <http://asintapi.lslima.me>

### 3.1.3 Client

I wanted to use a modern javascript framework to develop the user interface. Although there are many frameworks and libraries available, *React* seemed to be picking up momentum in the open source community and the fact that it was made by *Facebook* got my curiosity. The chosen stack is described below:

- React: A view library designed by *Facebook* for building complex user interfaces <https://facebook.github.io/react/>. React is normally tied together with a Flux implementation. Flux was designed as an alternative to MVC with it's unidirectional data flow <https://facebook.github.io/flux/>.
- Redux: The chosen flux implementation based on it's simplicity and popularity <http://rackt.org/redux/index.html>. It's a predictable state container for javascript apps and has some interesting features such as the possibility to redo or undo actions accumulated for debugging purposes when used together with *ReduxDevTools*.
- Webpack: Very popular and practical module bundler that transforms modules with dependencies into static assets. <https://webpack.github.io/>

The learning curve for this stack is a bit steep, especially when it comes to *Webpack's* configuration. After the initial troubles, we can make use of some modern features such as hot-reload with time travel (presented at React-Europe 2015 <https://www.youtube.com/watch?v=xsSn0QynTHs>). React is interesting because it enforces programmers to write code in a modular way and abandons the separation of code philosophy that other frameworks adhere to, which may cause some time to get used to (for example, we can use *jsx*, which is similar to html inside a .js file). Contrary to the norm, in most cases, *React* doesn't manipulate the DOM directly, instead it creates an in-memory representation of it, called virtual DOM (see <https://auth0.com/blog/2015/11/20/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/>).

## 3.2 Scaling the application

### 3.2.1 Environment used

In order to better understand how PAAS work, I decided to implement my own solution. Containers have been the backbone for PAAS solutions for



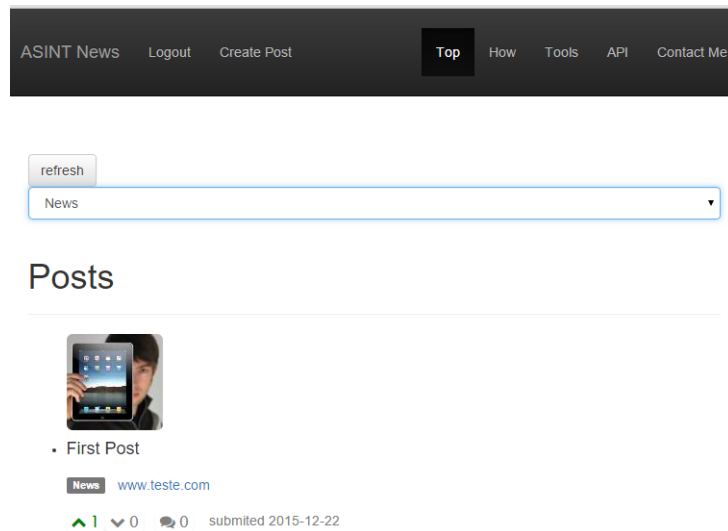


Figure 3: Client interface - <http://client.lslima.me>

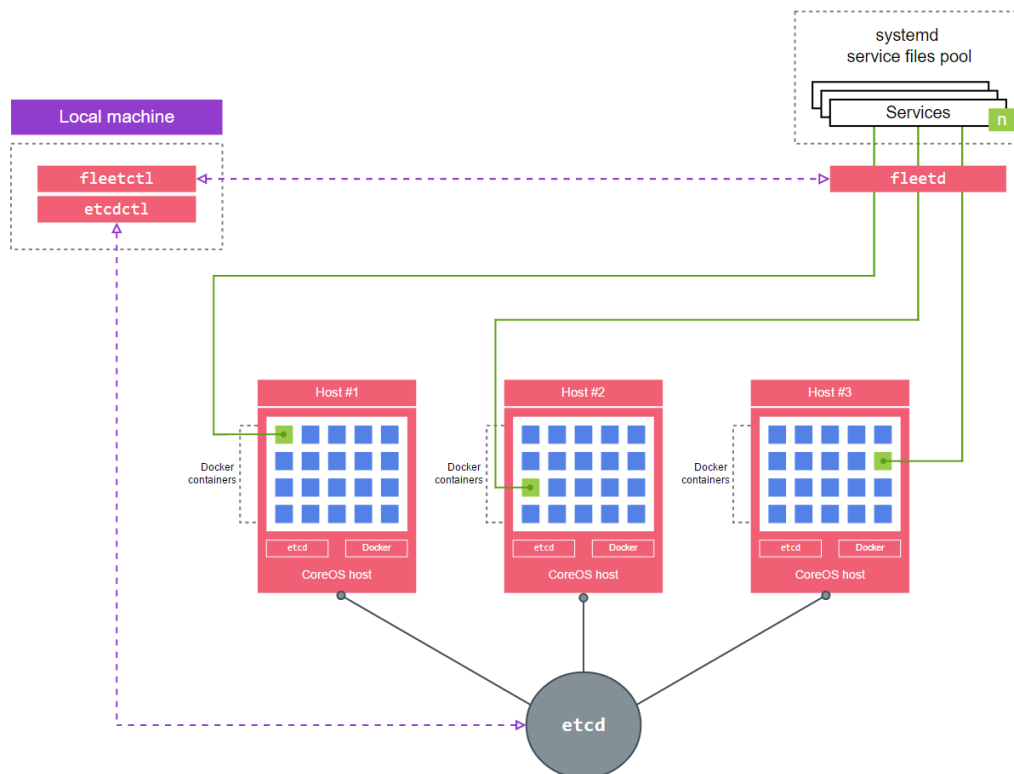


Figure 4: A high-level illustration of the CoreOS cluster architecture

years. For example, Google starts over 2 billion containers per week.<sup>1</sup>, as a result I turned my attention to Docker - “an open source container orchestration engine that separates applications from their underlying operating system” <https://www.docker.com/>. This application will use multiple containers that can easily be replicated, deployed and distributed.

I picked an open source stack to manage docker containers using *CoreOs*, *Fleet*, *Flannel* and *etcd*. The *CoreOs* website does a good job explaining these tools, so I will only present a small overview for each one:

- **CoreOs** - is a stripped down *Linux* with some *PAAS* like features. It's a good environment for running our *Docker* containers, each one, running only the dependencies required for it to run. CoreOs doesn't come with a package manager, and as a result, our programs have to run within containers. To manage the containers, *CoreOs* uses *Systemd* and *Fleet*.
- **Fleet** - makes possible to treat our *CoreOs* cluster as having a single *init* system. *fleet* makes use of *systemd* unit files for scheduling containers across our cluster. If a machine fails, *fleet* can automatically schedule the containers running on that machine into another one in the cluster. By installing *fleetctl* in our host computer, we can easily access a machine running a container using SSH. For more information and examples, see <https://coreos.com/using-coreos/clustering/>.
- **Flannel** - makes possible for containers to communicate with each other, by giving each container an IP address internal to the cluster. The network details necessary for translating the virtual IPs are stored into *etcd*. Read more about *Flannel* here <https://github.com/coreos/flannel>.
- **etcd** - is a key value distributed store. Any machine in a cluster can easily access this store using HTTP methods to register or get connection details from other machines in the cluster. One common use case for *etcd* is to register a set of machines running copies of an application. A load balancer can then watch *etcd* for changes in the connection details for those machines, and react to those changes. For more information check <https://coreos.com/etcd/>.

I have a cluster of three CoreOS machines running on DigitalOcean. There is also another machine outside the cluster, running a public facing *Nginx* proxy, that automatically gives each service a subdomain.

---

<sup>1</sup><http://anandmanisankar.com/posts/container-docker-PaaS-microservices/>

We start by creating our three CoreOs machines using a *cloud config* *yaml* file. Every machine inside the cluster uses the same configuration file and the same *etcd* discovery link <https://discovery.etcd.io/b5428f75fdcefc0375366b0bd4059aa7>. This link was generated by accessing <https://discovery.etcd.io/new?size=3>. Here the “size=3” represents the number of *etcd* replicas to be created, in this case, every machine has a copy of the *etcd* store. In the same configuration file, we need to tell which services we want to be running on our machines. In this case, we need *fleet*, *flannel* and of course *etcd*. Figure 5 shows the information relative to our cluster.

```
{
  action: "get",
- node: {
    key: "/_etcd/registry/b5428f75fdcefc0375366b0bd4059aa7",
    dir: true,
    - nodes: [
      - {
        key: "/_etcd/registry/b5428f75fdcefc0375366b0bd4059aa7/97198d8b09c5501",
        value: "463d75706c0e4d09881d108dc43d84d7=http://10.131.113.84:2380",
        modifiedIndex: 937451334,
        createdIndex: 937451334
      },
      - {
        key: "/_etcd/registry/b5428f75fdcefc0375366b0bd4059aa7/b1d4e105524215a0",
        value: "cb9c34b841ff4802b961855278acd896=http://10.131.112.77:2380",
        modifiedIndex: 937451245,
        createdIndex: 937451245
      },
      - {
        key: "/_etcd/registry/b5428f75fdcefc0375366b0bd4059aa7/76c3254a0b5f0704",
        value: "a9d7544cf6b24c6996bb8aaad0ef3f3e=http://10.131.111.70:2380",
        modifiedIndex: 937451318,
        createdIndex: 937451318
      }
    ],
    modifiedIndex: 937448471,
    createdIndex: 937448471
  }
}
```

Figure 5: etcd discovery link: <https://discovery.etcd.io/b5428f75fdcefc0375366b0bd4059aa7>

We can list our cluster machines by using “fleetctl list-machines” on our host machine. Now that we have our machines running, we can use *fleet* to schedule our containerized services.

### 3.2.2 Services

- **RethinkDb cluster** - I’m running *RethinkDb* in cluster mode on 3 machines. The tables used are “posts”, “users” and “comments”. Each

```
osboxes@osboxes ~/production-asint $ fleetctl list-machines
MACHINE      IP            METADATA
463d7570...  46.101.1.193  -
a9d7544c...  46.101.72.148 -
cb9c34b8...  46.101.84.98  -
```

Figure 6: We can use *fleetctl* on our host machine to list all the machines belonging to the cluster.

table is *sharded* 3 times, meaning, each machine has a primary table containing one third of the original table. This allows for faster queries, since *RethinkDb* can query the *sharded* data in parallel. However, *sharding* comes at a cost. If one machine fails, a portion of our data is lost. To solve this, each *shard* has 3 replicas, distributed through our machines. Now, if something happens to our data, *RethinkDb* has the necessary tools to automatically recover from it, if certain conditions are met.<sup>2</sup> Note that the number of *shards* and replicas can be chosen by the user.




Servers connected to the cluster		
 <b>Fire</b> default	3 primaries, 6 secondaries	<b>b5efbdb51049</b> , up for a day
 <b>Water</b> default	3 primaries, 6 secondaries	<b>4cf553dc8a8f</b> , up for 9 hours
 <b>Earth</b> default	3 primaries, 6 secondaries	<b>851785ee9d00</b> , up for a day

Figure 7: RethinkDb Cluster

- **Web-Server** - I'm running the web-server on two machines. These instances are registered within the shared *etcd* store. Then, a load balancer can access this *etcd* to know the connection details of the two instances. This is particularly useful for failover purposes. If one of these two instances dies, *fleet* will automatically schedule a new one into the third machine, updating the *etcd* store with the new connection details for the load balancer to pick up.
- **Nginx Load Balancer** - For load balance, the third machine is running an *Nginx* instance. Since our application makes use of web-sockets, we have to make sure each client is always communicating with the same server instance, otherwise, the web-socket connections

<sup>2</sup><https://rethinkdb.com/docs/failover/>.

Servers used by this table		
Shard 1	~0 documents	
Earth		Primary replica
Fire		Secondary replica
Water		Secondary replica
Shard 2	~0 documents	
Fire		Primary replica
Water		Secondary replica
Earth		Secondary replica
Shard 3	~0 documents	
Water		Primary replica
Fire		Secondary replica
Earth		Secondary replica

Figure 8: Table “users” sharded across 3 machines

break. For this purpose, *Nginx* has an option called “ip\_hash” that can be defined inside the upstream configuration section. There is also another change that we have to make to allow *Nginx* to forward the web-socket protocol. This is all described on the *socket.io* documentation <http://socket.io/docs/using-multiple-nodes/>.

- **Redis** - Now that we have the web-socket connections working on multiple nodes, we have to make sure clients from different server nodes will receive the changes accordingly. To solve this, we have to make use of a session store, shared by the two servers. Then, we can make use of the official *socket.io-redis* adapter in our server code, allowing the servers to broadcast and emit events to and from each other.
- **Public Facing Nginx Proxy** - If something happens to our intra-cluster load balance instance, a new instance will automatically get scheduled on a working machine inside the cluster, with a new IP address and port. To solve this issue, a public facing *nginx* proxy is running in a separate machine outside the cluster alongside with a new *etcd* store. This *etcd* store has a public IP address<sup>3</sup> that can be used by our containers inside the cluster to register themselves, whenever they spawn, using a simple HTTP PUT request. For example, the load balancer instance will have a fixed subdomain name with the public

<sup>3</sup>in a real configuration we should setup IPTables accordingly for security purposes.

IP address of the load balancer instance that is running on that time. The *nginx* proxy can simply watch the external *etcd* store for changes and reload itself with the new configuration. Ideally we would use two public proxies for high availability, sharing the same virtual IP, and if one of them crashes, the other can pick up the requests automatically. DigitalOcean provides this functionality since October of this year and they call it *Floating IPs* (Amazon AWS calls it *Elastic IP*. For more information on *Floating IPs* check <https://www.digitalocean.com/company/blog/floating-ips-start-architecting-your-applications-for-high-ava>

```
osboxes@osboxes ~/production-asint $ fleetctl list-units
```

UNIT	MACHINE	ACTIVE	SUB
api-discovery@1.service	463d7570.../46.101.1.193	active	running
api-discovery@2.service	a9d7544c.../46.101.72.148	active	running
apidocs.service	a9d7544c.../46.101.72.148	active	running
asintapi-lb.service	cb9c34b8.../46.101.84.98	active	running
asintapi@1.service	463d7570.../46.101.1.193	active	running
asintapi@2.service	a9d7544c.../46.101.72.148	active	running
asintproj@1.service	a9d7544c.../46.101.72.148	active	running
client-discovery@1.service	a9d7544c.../46.101.72.148	active	running
redis-discovery@1.service	463d7570.../46.101.1.193	active	running
redis-discovery@2.service	cb9c34b8.../46.101.84.98	active	running
redis@1.service	463d7570.../46.101.1.193	active	running
redis@2.service	cb9c34b8.../46.101.84.98	active	running
rethinkdb-discovery@1.service	463d7570.../46.101.1.193	active	running
rethinkdb-discovery@2.service	a9d7544c.../46.101.72.148	active	running
rethinkdb-discovery@3.service	cb9c34b8.../46.101.84.98	active	running
rethinkdb@1.service	463d7570.../46.101.1.193	active	running
rethinkdb@2.service	a9d7544c.../46.101.72.148	active	running
rethinkdb@3.service	cb9c34b8.../46.101.84.98	active	running

Figure 9: Services running in the cluster

### 3.2.3 Service Registration and Load Balancing the two Web-Server instances

Figure 10 shows the registered containers present on the shared *etcd* store inside the cluster. For some instances, we can see a sidekick process used for service registration. The load balancer instance, responsible for load balancing the two web-server instances, watches the intra-cluster *etcd* store for changes to both web-server instances and reloads itself if any one of them is scheduled into another machine.

```
core@coreos1 ~ $ etcdctl ls --recursive
/announce
/announce/services
/announce/services/rethinkdb1
/announce/services/rethinkdb3
/announce/services/rethinkdb2
/services
/services/redis
/services/redis/upstream
/services/redis/upstream/redis2
/services/redis/upstream/redis1
/services/asintapi
/services/asintapi/upstream
/services/asintapi/upstream/asintapi2
/services/asintapi/upstream/asintapi1
/services/asintproj
/services/asintproj/upstream
/services/asintproj/upstream/asintproj1
```

Figure 10: Services registered within etcd.

```
core@coreos1 ~ $ etcdctl get /services/asintapi/upstream/asintapi1
10.1.7.24:4000
```

Figure 11: Running etcdctl get it is possible to obtain our services' IP address and port.

```
upstream app {
    ip_hash;
    {{range getvs "/services/appname/upstream/*"}}
        server {{.}};
    {{end}}
}
```

Figure 12: Upstream configuration section of the nginx load balancer instance. Each client's IP is guaranteed to always talk with to same server. The list of servers to load balance is simply the registered web-servers instances within etcd.

### 3.2.4 DNS Registration, Subdomains and High Availability Proxy

An *Nginx* instance running as a proxy is going to watch the public *etcd* store for changes on the configuration details of the three public services registered:

1. client (asintproj1).
2. load balancer instance (asintapi-lb), responsible for load balancing both web-server/API (asintapi) instances.

### 3. API documentation (apidocs).

```
{
  action: "get",
  - node: {
    dir: true,
    - nodes: [
      - {
        key: "/subdomains",
        dir: true,
        - nodes: [
          - {
            key: "/subdomains/client",
            value: "46.101.72.148:3000",
            modifiedIndex: 22,
            createdIndex: 22
          },
          - {
            key: "/subdomains/asintapi",
            value: "46.101.84.98:32777",
            modifiedIndex: 3,
            createdIndex: 3
          },
          - {
            key: "/subdomains/docs",
            value: "46.101.72.148:32796",
            modifiedIndex: 19,
            createdIndex: 19
          }
        ],
        modifiedIndex: 3,
        createdIndex: 3
      }
    ]
  }
}
```

Figure 13: <http://46.101.75.177:2379/v2/keys?recursive=true> - The *etcd* store outside the cluster is used to register the web-server's load balancer instance, the API documentation web-page and the client web-page. Each one of these instances have a subdomain that doesn't change, and whose IP address is automatically updated even if they are scheduled into other machines.



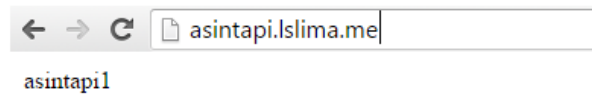


Figure 14: Root endpoint of the API printing which server instance the client is connected to.

Listing 3: Nginx configuration, using Golang's text/template engine, to generate the virtual hosts (server blocks) for each subdomain

---

```
{{range gets "/subdomains/*"}}
server {
    server_name {{base .Key}}.lslima.me;
    location / {
        proxy_pass http://{{.Value}};
        proxy_redirect off;
    }
}
{{end}}
```

---

### 3.2.5 High Level View of the Architecture

Figure 15 shows a high level view of the architecture.

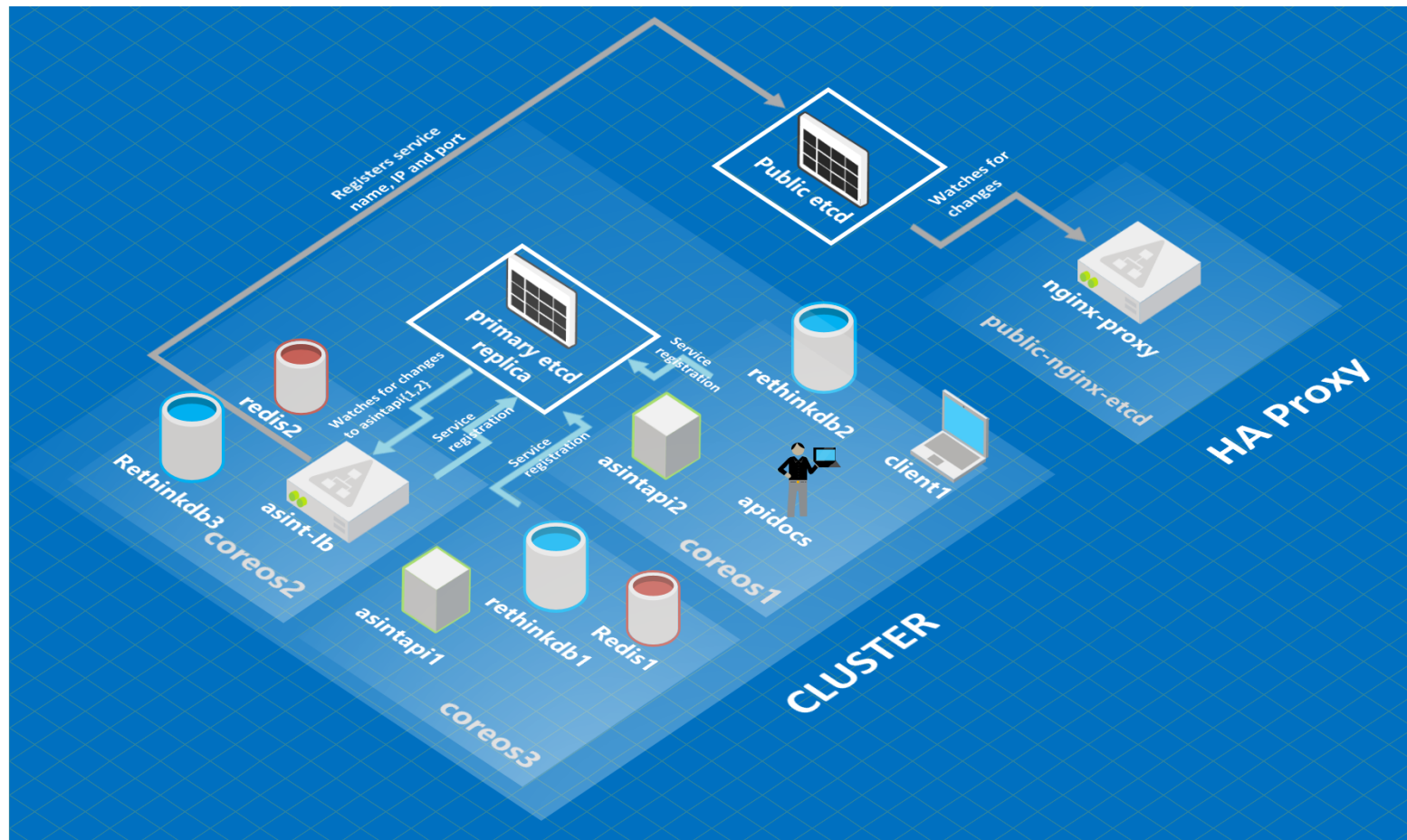


Figure 15: High level view of the architecture

## 4 Appendix

### 4.1 Helpful links

- How to create a *CoreOs* cluster on digitalocean - <https://www.digitalocean.com/community/tutorials/how-to-set-up-a-coreos-cluster-on-digitalocean>.
- RethinkDb and CoreOs - <http://blog.justonepixel.com/geek/2014/10/03/rethinkdb-and-coreos/>
- Using JSON Web Tokens as API keys - <https://auth0.com/blog/2014/12/02/using-json-web-tokens-as-api-keys/>
- Anatomy of a JSON Web Token - <https://scotch.io/tutorials/the-anatomy-of-a-json-web-token>.
- REST+JSON API Design - Best Practices for Developers - <https://www.youtube.com/watch?v=hdSrT4yjS1g>
- Scaling Real-Time applications using Redis - <https://github.com/rajaraodv/redispubsub>.
- Playlist containing an intro to *Docker* and a five video series on how to scale docker using CoreOs <https://www.youtube.com/watch?v=pGYAg7TMmp0&list=PLoYCgNOIyGAAzevEST2qm2Xbe3aeLFvLc>.

### 4.2 DigitalOcean machines




Droplets				Search By Droplet Name
Img	Name	IP Address	Created ▲	
	<b>coreos3</b> 512 MB Memory / 20 GB Disk / LON1	46.101.1193	5 hours ago	<a href="#">More ▼</a>
	<b>coreos2</b> 512 MB Memory / 20 GB Disk / LON1	46.101.84.98	5 hours ago	<a href="#">More ▼</a>
	<b>coreos1</b> 512 MB Memory / 20 GB Disk / LON1	46.101.72.148	5 hours ago	<a href="#">More ▼</a>

Figure 16: 3-machine cluster running on DigitalOcean


Img	Name	IP Address	Created
	public-nginx-etcdd 512 MB Memory / 20 GB Disk / LON1	46.101.75.177	4 days ago <a href="#">More</a>

Figure 17: Public facing Nginx proxy running outside the cluster with it's own etcd store for domain registration

### 4.3 DNS configuration

A	@	46.101.75.177	Save	Remove
CNAME	*	lslima.me.	Save	Remove
CNAME	asintapi	lslima.me.	Save	Remove
CNAME	docs	lslima.me.	Save	Remove
CNAME	client	lslima.me.	Save	Remove
NS	ns1.digitalocean.com.		Save	Remove
NS	ns2.digitalocean.com.		Save	Remove
NS	ns3.digitalocean.com.		Save	Remove

#### Zone File

```
$ORIGIN lslima.me.
$TTL 1800
lslima.me. IN SOA ns1.digitalocean.com. hostmaster.lslima.me. 1451755893 10800 3600 604800 1800
lslima.me. 1800 IN NS ns1.digitalocean.com.
lslima.me. 1800 IN NS ns2.digitalocean.com.
lslima.me. 1800 IN NS ns3.digitalocean.com.
lslima.me. 1800 IN A 46.101.75.177
*.lslima.me. 1800 IN CNAME lslima.me.
asintapi.lslima.me. 1800 IN CNAME lslima.me.
docs.lslima.me. 1800 IN CNAME lslima.me.
client.lslima.me. 1800 IN CNAME lslima.me.
```

Figure 18: DNS configuration on my DigitalOcean account

## 4.4 Fleet Files (only a subset)

Listing 4: asintapi@.service

---

```
[Unit]
Description=asintapi%i
After=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=/usr/bin/docker kill asintapi%i
ExecStartPre=/usr/bin/docker rm asintapi%i
ExecStartPre=/usr/bin/docker pull lslima/asintapi
ExecStart=/usr/bin/docker run -p 4000:4000 -e APPNAME=asintapi%i
    --name asintapi%i -P lslima/asintapi
ExecStop=/usr/bin/docker kill asintapi%i

[X-Fleet]
Conflicts=asintapi*
```

---

This file acts as a template for our web-server service. In order to access the name and id of the instance we declare an environment variable called APPNAME. The web-server application is going to render this variable name on it's root endpoint (see Figure 14). This is simply used for debugging purposes.

The command "-P lslima/asintapi" simply pulls the previously generated docker image containing the application, from my personal docker repository. The X-Fleet section, contains a "Conflicts" declaration, that prevents two instances that contain the name "asintapi", from being scheduled on the same machine.

Listing 5: submitting template and starting a service

---

```
fleetctl submit asintapi@
fleetctl start asintapi@{1..2}
```

---

We submit the template to the cluster and then schedule two instances of the service.

Listing 6: api-discovery@.service - sidekick service to register asintapi@.service

---

```
[Unit]
Description=Announce asintapi%i
BindsTo=asintapi@%i.service
```

---

```
After=asintapi@%i.service
```

```
[Service]
```

```
ExecStart=/bin/sh -c "sleep 10; while true; do etcdctl set  
    /services/asintapi/upstream/asintapi%i \"$(sleep 5 && docker  
    inspect -f '{{.NetworkSettings.IPAddress}}' asintapi%i):4000\"  
    --ttl 60;sleep 45;done"
```

```
ExecStop=/usr/bin/etcdctl rm /services/asintapi/upstream/asintapi%i
```

```
[X-Fleet]
```

```
MachineOf=asintapi@%i.service
```

---

This file tells fleet to bind this service to the same machine that its companion service `asintapi@.service` is running. The command inside `ExecStart` registers the corresponding docker's container IP in the etcd store. We launch it the same way as in Figure 5

Listing 7: `asintapi-lb.service`

---

```
[Unit]
```

```
Description=asintapi-lb
```

```
After=docker.service
```

```
[Service]
```

```
TimeoutStartSec=0
```

```
ExecStartPre=/usr/bin/docker kill asintapi-lb
```

```
ExecStartPre=/usr/bin/docker rm asintapi-lb
```

```
ExecStartPre=/usr/bin/docker pull lslima/stickylb
```

```
ExecStart=/usr/bin/sh -c "/usr/bin/docker run -e  
    SERVICE_NAME=asintapi -e ETCD=\"$(ifconfig docker0 | awk  
    '/<inet>/ { print $2}'):2379\" -P --name asintapi-lb  
    lslima/stickylb"
```

```
ExecStartPost=/usr/bin/sh -c "sleep 3 && curl -X PUT  
    46.101.75.177:2379/v2/keys/subdomains/asintapi -d value=$(curl  
    -s checkip.dyndns.org | sed -e 's/.*Current IP Address: //' -e  
    's/<.*$//'):$(/usr/bin/docker ps | grep 'asintapi-lb' | grep -o  
    '[0-9]\+-->' | grep -o '[0-9]\+')"
```

```
ExecStop=/usr/bin/docker stop asintapi-lb
```

---

This file tells fleet to run the load balancer application. Then, after the service starts, we execute the command inside `ExecStartPost`, in order to register this instance on the public etcd running on a machine outside the cluster with the IP address "46.101.75.177". The grep commands are simply to parse the correct IP and port.

For more details read <https://coreos.com/fleet/docs/latest/unit-files-and-scheduling.html>.

## 4.5 Other Considerations

**REST API** - To create the REST API I tried to follow a set of rules:

- The endpoints should refer to resources and not verbs.
- Don't attribute behavior to the endpoint definition. We should give the client the full representation of the resource and let them determine what they want to do with them.
- POST, GET, PUT and DELETE don't have a one to one relationship with CRUD operations.
- For creating resources we use the method PUT, which is an idempotent method. We must supply all the necessary data for the request because according to the HTTP specification, calling one time or multiple times this operation the result should be the same. For example, in my case, to create a comment I use `"/comments/postId"` because the client knows the post ID ahead of time and all the information necessary for a comment creation is required.
- In cases where we need to create a resource but we don't know the location of where it is going to be created, we can use the POST method. In my case, to create a parent resource, such as a news "Post" the client calls the `"/posts"` endpoint with the method POST and receives as response the HTTP code 201 (created) with the url location of the resource created.
- To update a resource the HTTP method POST is used, which according to the HTTP specification is not an Idempotent method and allows for partial updates. This is the case for the operations upvote/downvote on my API.

## 4.6 Other tools used

**Developing Tools - client side:**

1. Eslint: Javascript linting tool.

2. Babel: Babel allows to use all the features that ES6 (latest official javascript version) brings today, without sacrificing backwards compatibility for older browsers. <https://babeljs.io/>
3. ReduxDevTools